# Managing Large-scale System Deployment and Configuration with Windows PowerShell

Scott Hanselman, Chief Architect, Corillian Corporation

## Software Used

- Windows PowerShell 1.0
- Visual Studio 2003 and 2005
- Windows Server 2003 R2
- SQL Server 2005
- CruiseControl.NET
- MSBUILD and NANT
- NCover for Code Coverage
- NUnit for Unit Testing
- SubVersion and AnkhSVN's libraries

## Introduction

Corillian Corporation is an eFinance vendor that sells software to banks, credit unions and financial institutions all over the world to run their online banking sites. If you log into a bank in the United States to check your balances, account history or pay a ball, there's a 1 in 4 chance you're talking to a system built by Corillian, running on Windows, and based on .NET. Our core application is called Voyager. Voyager is a large managed C++ application that acts as a component container for banking transactions and fronts a financial institution's host system, typically a mainframe, provides a number of horizontal services like scalability, audit-ability, session state management as well as the ability to scale to tens of thousands of concurrent online users.

We have a dozen or so applications that sit on top of the core Voyager application server, like Consumer Banking, Corporate Banking, eStatements, and Alerts. These applications are usually Web applications and many expose Web Services. These applications, along with Voyager are deployed in large web farms that might have as few as five computers working as a unit, or as many as 30 or more.

While Voyager is now written and compiled as a managed C++ application, all the applications that orbit Voyager and make up the product suite are written in C#. The core Voyager application server is ten years old now and like many large Enterprise systems, it requires a great deal of system-level configuration information to go into production. Additionally, the sheer number of settings and configuration options are difficult to manage when viewed in the context of a mature software deployment lifecycle that moves from development to testing to staging to production.

## What is Configuration?

Configuration might be looked upon as anything that happens to a Windows system after the base operating system has been installed. Our system engineers spend a great deal of time installing and configuring software, but more importantly they spend time managing and auditing the configuration of systems. What was installed, when was it installed, and are all of these servers running the same versions of the same software? Maintaining configuration is equally or more important as applying configuration. Our aim was to make software deployment easier, much faster, and make ongoing maintenance a "no touch" prospect.

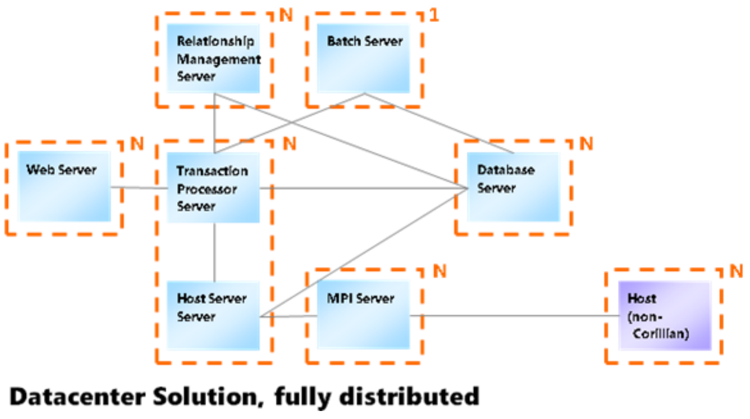

Relationship Management Server
N
Batch Server
1
Web Server
N
Transaction Processor Server
N
Database Server
N
Host Server Server
MPI Server
N
Host (non-Corillian)
N
Datacenter Solution, fully distributed

Figure 1 - Voyager in a Production Environment

Configuration takes on many forms in large Windows-based systems. Some examples of system-level configuration are DOM settings, keys in the Registry, the IIS Metabase settings, and .config settings stored in XML. There are business level configuration settings stored in the database, there are multilingual resources stored in XML RESX files, and there are assets like images and other files that are stored on the web server. Configuration can also take the form of client specific markup within an ASPX page such as configuration of columns in a grid that could be set at design time rather than configured at runtime. Configuration can also include endpoint details, like IP addresses and SSL certificates, or host (mainframe) connectivity information.

Large enterprise applications of any kind, in any industry, written in any technology, are typically non-trivial to deploy. Voyager also allows for "multi-tenant" configuration that lets us host multiple banks on a single running instance of the platform, but this multiplies the number of configuration options, increases complexity and introduces issues of configuration scope.

When hosting a number of financial institutions on a single instance of Voyager we have to keep track of settings that affect all financial institutions vs. settings scoped to a single FI in order to meet service level agreements as well as prevent collisions of configuration.

Each instance of Voyager can run an unlimited number of financial institutions, each partitioned into their own space, but sharing the same hardware. We've picked the arbitrary number of fifty FIs and called one instance a "*Pod*." We can run as many pods as we like in our hosting center, with Voyager itself as the only shared software, so each pod could run a different version of Voyager, which each FI selects from a menu of applications. Each runs a different version of their custom banking platform, like Retail Banking or Business Banking.

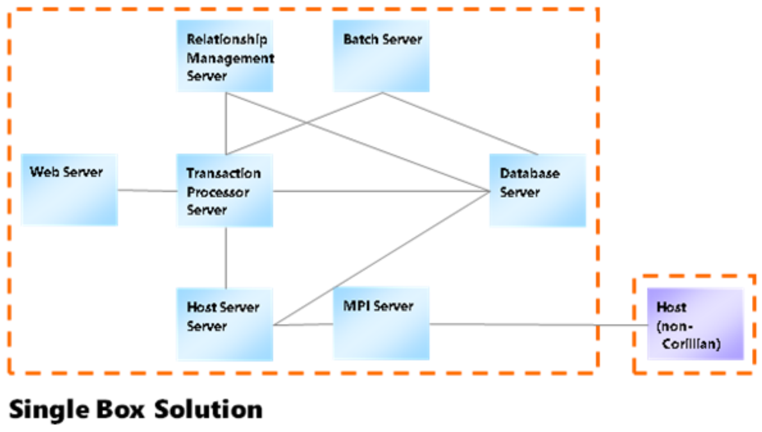


Figure 2 - Voyager running on a VM or Demo Machine

Some classes of configuration items like IIS settings are configured on a per-virtual-directory basis and map one to one to a bank or financial institution while some settings are shared amongst all financial institutions.  Additionally, changes to some configuration settings are recognized immediately by the system, while other more drastic settings might be recognized only after an AppDomain or application restart.

## Representing Configuration Settings

We've chosen to roll up the concept of configuration into a file per financial institution stored in xml and one more file for the containing instance of the Voyager application server, each with an associated schema. These files are not meant to be edited directly by a human.

We partitioned settings by scope, by type, by effect (immediate versus scheduled) and by instance of our Voyager application server – that is, we scope configuration data by *Pod* and by *Bank*. One *Hosting Center* can have many *Pods*. One Pod has many Banks, and a Pod might be installed in any number of *Environments* like Development, Staging, or Production.

So far these are all logical constructs – not physical. The underlying platform is very flexible and mapping these services to a physical layout might find the system running fully distributed in a data center as in Figure 1 or the entire suite of Applications running on a single virtual machine in Figure 2. This means that a single physical server might fill a different *Role* like *Web Server* or *Database Server*. The pod configuration file maintains a list of the ID of each computer in the pod and the roles that computer takes on. For example, the simple pod configuration file below has two environments, staging and production, and each environment has just one machine, one for staging and one for production. Each of these machines is in a number of roles, playing web server, database server and transaction processor in a configuration similar to Figure 2. This format is simple and flexible enough to keep an inventory of a configuration composed of *n* number of machines as in Figure 1.

```
<PodSettings [namespaces removed for clarity]>
      <environments>
            <environment name="staging">
                  <servers>
                        <server>
                              <id>192.168.1.2</id>
                              <roles>
                                    <role>tp</role>
                                    <role>web</role>
                                    <role>rm</role>
                                    <role>sql</role>
                              </roles>
                        </server>
                  </servers>
            </environment>
            <environment name="production">
                  <servers>
                        <server>
                              <id>192.168.1.1</id>
                              <roles>
                                    <role>tp</role>
                                    <role>web</role>
                                    <role>rm</role>
                                    <role>sql</role>
                              </roles>
                        </server>
                  </servers>
            </environment>
      </environments>
          ...Other Pod Settings here...
</PodSettings>
```
Figure 3 - A simple Pod Settings XML file

Each pod has only one PodSettings.xml file as these settings are global to the pod in scope. Each financial institution has a much more complex settings.xml file that contains all settings across all applications they've purchased that they might want to manage. We've found the natural hierarchy of XML along with namespaces and its inherent extensibility as a meta-language to be much easier to deal with versus a database. Storing all the settings in a file on a per-FI basis also has a very specific benefit to our vertical market as well as making that file – the authoritative source for their settings – easier to version.

## Our Solution

The solution needed to address not only the deployment of software, but the configuration of software, specifically the *ongoing reconfiguration* that occurs through the life of a solution. Taking a machine from a fresh OS install to a final deployment was an important step, but we also needed to manage the state of the system in production. In our case, banks want to make changes not only to text, and look and feel, but also business rules within specific applications. These changes need to be audited, some applied immediately and some applied on a schedule. Each needs to be attached to an individual who is accountable for the change.

These requirements pointed us in the direction of a version control system, specifically Subversion. Subversion manages all changes to the file system, that is, anything from code in the form of assemblies, to markup. All *configuration*, as defined above, is stored in a financial institution-specific XML file and is versioned along with every other file in the solution. It's significant to point out that we are versioning the actual deployed solution, not the source code. The source code is in a different source code repository, and managed in the traditional fashion; this Subversion system manages the application in its deployed, production state.

There are many servers in a deployed production system – upwards of dozens – and they will each run a custom Agent Service that hosts PowerShell Runspaces enabling the complete remote administration of the system using a single open TCP port.  Rather than *pushing* software for deployment to these many remote systems, imperative commands are sent to these remote agents and the remote systems *pull* their assigned deployments from a version within Subversion. After *deployment* – the laying down of bits on the disk – a *publish* occurs, and PowerShell scripts spin through the settings XML file for the particular financial institution and each class of setting, for the registry, database, config file, etc., is applied to the solution.

When a FI wants to make a change to their system, they log into a secure SharePoint extranet and edit the configuration of their solution in a user interface that was code-generated using their own settings

XML file as the source. Their changes are applied not to the production system, but rather to the settings XML file stored in subversion. Settings can be applied immediately or on a schedule, depending on the ramifications of a particular settings change. Settings that require a restart of IIS or another service will happen on a scheduled basis, while look and feel changes can happen immediately.  Changes are sent to Subversion and versioned along with the identity of the requesting user for audit and potential rollback purposes. These changes can be double checked by a Customer Service Representative if need be. Assuming the changes are valid, they are then *pulled* down by the remote agents and published to the deployed solution. Rollbacks or "reassertion" of settings is performed in the identical fashion using an earlier version of the deployed solution and settings.

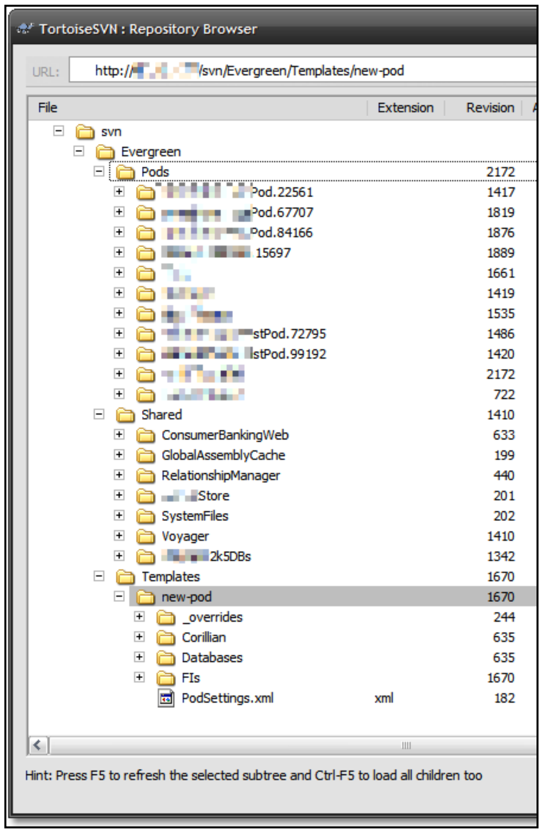
PowerShell scripts handle both the deployment of software and publishing of settings. PowerShell commands are used not only at the command-line, but also hosted within ASP.NET applications and MMC/WinForms applications, and remotely via a custom host. PowerShell interacts with Subversion, and nearly every possible kind of object on the system that requires configuration.

## Storing Applications and Configuration

Voyager is just the base of the pyramid of a much larger suite of applications that a bank might choose. Each web application might have configuration data stored separately or shared with other applications. Here is a list of different potential bits of configuration that could be "set":

- Application Settings
  - Assemblies, code and files on disk
  - System DLLs, prerequisites
  - GAC'ed Assemblies
  - DCOM/COM+ settings and permissions
  - Registry Settings
  - File System ACLs (Access Control Lists) and Permissions
  - XML configuration files
  - Settings stored in a Database
  - Mainframe/Host Connectivity Details
- Web Applications
  - Web Server (IIS Metabase) Settings
  - Web Markup (ASPX)
  - Stylesheets (CSS)
  - Multilingual Resources (RESX)
  - Asset management (Graphics, Logos, Legal Text, etc)

Everything in this list, and more, is applied on every single machine in an application farm once the operating system has been installed.  We'll talk more about how Applications are deployed, then how configuration is published to those applications after deployment.



```
TortoiseSVN : Repository Browser
URL: http://[redacted]/svn/Evergreen/Templates/new-pod

File                                  Extension   Revision
⊟ svn
  ⊟ Evergreen
    ⊟ Pods                                          2172
      ⊞ [redacted]Pod.22561                         1417
      ⊞ [redacted]Pod.67707                         1819
      ⊞ [redacted]Pod.84166                         1876
      ⊞ [redacted] 15697                            1889
      ⊞ [redacted]                                  1661
      ⊞ [redacted]                                  1419
      ⊞ [redacted]                                  1535
      ⊞ [redacted]stPod.72795                       1486
      ⊞ [redacted]lstPod.99192                      1420
      ⊞ [redacted]                                  2172
      ⊞ [redacted]                                   722
    ⊟ Shared                                        1410
      ⊞ ConsumerBankingWeb                           633
      ⊞ GlobalAssemblyCache                          199
      ⊞ RelationshipManager                          440
      ⊞ [redacted]Store                              201
      ⊞ SystemFiles                                  202
      ⊞ Voyager                                     1410
      ⊞ [redacted]2k5DBs                            1342
    ⊟ Templates                                     1670
      ⊟ new-pod                                     1670
        ⊞ _overrides                                 244
        ⊞ Corillian                                  635
        ⊞ Databases                                  635
        ⊞ FIs                                       1670
          PodSettings.xml                 xml        182

Hint: Press F5 to refresh the selected subtree and Ctrl-F5 to load all children too
```

**Figure 4 - A Subversion Repository showing a partial Suite of Applications in a Pod**

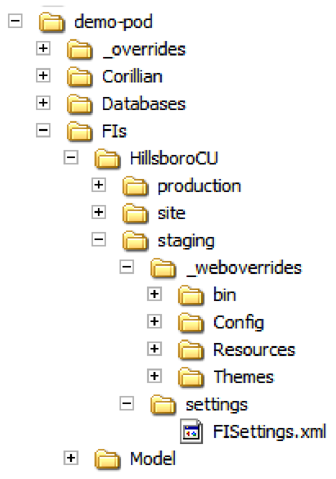
## Storing Applications in their Deployed State

Before configuration settings can be applied the actual applications must be installed. Previously most of our applications were installed using MSI installer technology, but as the suite of applications grew, so did the number of installers. As you know, MSI installers can be finicky, particularly with regards to installation order. Patching is also problematic and often it's difficult to confirm that a farm of machines is all configured identically with the exact same version of an application. It's difficult to version an entire application farm.

We use Subversion as our source control system of choice, and appreciate its efficient use of disk space. We decided to use subversion to also store the binaries of deployed applications. Essentially, we "checked in our running application." This allows us to effectively version entire web farms running the suite. The snapshot of a repository is seen below in Figure 4.

The hierarchy presented by Subversion enables us to model the logical and aspects of the business and the physical aspects of the file system. The Templates folder contains a basic template for a new "model" Financial Institution. In Figure 5, named Pods contain FIs that contain environments. Beneath the environment are those folders that contain applications specific to a particular FI.

Shared applications like Voyager itself and System Files are versioned separately outside the Pod. Some FIs choose to customize their installations, and those customizations are always stored within that FI's pod.



```
⊟ demo-pod
  ⊞ _overrides
  ⊞ Corillian
  ⊞ Databases
  ⊟ FIs
    ⊟ HillsboroCU
      ⊞ production
      ⊞ site
      ⊟ staging
        ⊟ _weboverrides
          ⊞ bin
          ⊞ Config
          ⊞ Resources
          ⊞ Themes
        ⊟ settings
            FISettings.xml
    ⊞ Model
```

**Figure 5 - A Sample Pod and Credit Union showing multiple deployment Environments**

## In Search of a Versioned FileSystem - Subversion on top of NTFS

In this deployment model, we are using Subversion as a "versioned FileSystem" rather than as a source control system. This is largely a semantic distinction, but it has been a valuable one. The application deployment repository is kept separate from the source control repository and thought of differently.

We choose to use Subversion to manage our binaries rather than Windows 2003 Volume Shadow Copies service because of Subversion's transparency, as it's just a file system built on top of a file system. Also we valued its programmability and the transactional nature of Subversion's check-ins. We could have used SQL Server as a backing store, but Subversion required no licensing fees, was sufficiently transactional, and it already simply represented as a file system.

## Tying it all together with Windows PowerShell

Once the basic design for storing configuration was drafted, we found ourselves dreading writing all of this in C#. We felt the solution called for a scripting language that would allows us to "put the whole of Windows on a string." Since we'd be integrating.NET objects, legacy COM objects, manipulating the registry, and talking to databases, PowerShell seemed much more flexible than VBScript.

### Connecting PowerShell and Subversion

First we needed to integrate PowerShell and Subversion. While PowerShell supports the concept of a pluggable Drive Provider and there is the beginning of a Subversion Drive Provider for PowerShell[1], we decided to utilize a more explicit CRUD – Create, Read, Update, Delete –model for accessing files in Subversion.

One could use the TortoiseProc.exe application that is used by the TortoiseSVN tool that integrates subversion with Explorer with a script as seen below. It is an example of a decent pattern for calling tools that have discoverable paths (via the registry, etc) but aren't in the PATH proper.

```
if ($args.Length -lt 1) {
    write-host "usage: tsvn <command>"
    return
}

if ((test-path "HKLM:\Software\TortoiseSVN") -eq $false) {
 write-host -foregroundColor Red "Error: Could not find TortoiseProc.exe"
 return
}

$tortoiseKey = get-itemproperty "HKLM:\Software\TortoiseSVN"

if ($tortoiseKey -eq $null) {
 write-host -foregroundColor Red "Error: Could not find TortoiseProc.exe"
 return
}

$tortoise = $tortoiseKey.ProcPath

if ($tortoise -eq $null) {
 write-host -foregroundColor Red "Error: Could not find TortoiseProc.exe"
 return
}
```

[1] http://www.hanselman.com/blog/AnkhSVNAndAMonadSVNProvider.aspx

```
$commandLine = '/command:' + $args[0] + ' /notempfile /path:"' + ((get-
location).Path) + '"'
& $tortoise $commandLine
```

However, TortoiseProc is a standard command-line application and doesn't tightly integrate with PowerShell, leaving us to parse strings. Also, TortoiseProc is really meant to be used from TortoiseSVN and has poor username/password handling.

The most attractive option was to find a way to talk to Subversion directly using something in-process. We wanted a solution that integrated cleanly with PowerShell so that we could maximize reuse by caling our PowerShell scripts from not only the command line, but also from ASP.NET and a WinForms application. Arild Fines[2], the author of a popular Subversion Source Control Provider for Visual Studio.NET, provides a .NET library called NSvn.Core that fronts the "C" style API that is included with Subversion.  It doesn't appear to be distributed outside of Ankh, but it is distributed with the Ahkn install. The library has no documentation but it clearly shadows the Subversion API with some allowances and abstractions to make it easier to access from within .NET.

Here's an example script to get a file from Subversion to a local path.

```
param ([string]$svnurl       = $(read-host "Please specify the path to
SVN"),
       [string]$svnlocalpath = $(read-host "Please specify the local path")
      )

if ([System.IO.Path]::IsPathRooted($svnlocalpath) -eq $false)
{
  throw "Please specific a local absolute path"
}

[System.Reflection.Assembly]::LoadFrom((join-Path $GLOBAL:someGlobalPath -
childPath NSvn.Common.dll))
[System.Reflection.Assembly]::LoadFrom((join-Path $GLOBAL:someGlobalPath -
childPath NSvn.Core.dll))

$PRIVATE:svnclient = new-object NSvn.Core.Client
$PRIVATE:svnclient.AuthBaton.Add(
[NSvn.Core.AuthenticationProvider]::GetWindowsSimpleProvider() )
if ((test-Path $svnlocalpath) -eq $true )
{
  write-progress -status "Updating from $svnurl" -activity "Updating Working
Copy"
  $PRIVATE:svnclient.Update($svnlocalpath, [NSvn.Core.Revision]::Head,
$true)
}
else
{
  write-progress -status "Checkout from $svnurl" -activity "Updating Working
Copy"
```

[2] http://arildf.spaces.live.com/

```
  $PRIVATE:svnclient.Checkout($svnurl, $svnlocalpath,
[NSvn.Core.Revision]::Head, $true)
}
```

This script lets us checkout and update from Subversion, but there's no progress bar during the operation. The underlying NSvn .NET library will call a delegate to report its progress as an event, so the addition of these two lines after the creating of the NSvn.Core.Client object enables the script with a Progress bar. Notice that a script block is used as an anonymous method that is hooked into the Notification event of the NSvn Client object.

```
$PRIVATE:notificationcallback = [NSvn.Core.NotificationDelegate]{
        Write-Progress -status ("{0}: {1}" -f ($_.Action, $_.Path)) -
activity "Updating Working Copy"
    }

$PRIVATE:svnclient.add_Notification($notificationcallback
```

With these building blocks, we can easily talk to Subversion with PowerShell scripts, then reuse those scripts in different context within the overall solution, including ASP.NET, WinForms, or within UnitTests.

## Testing PowerShell with NUnit

We practice Continuous Integration at Corillian, running a Build Server for every project. Every time source is checked in, our Build Server – running an open source package called CruiseControl – checks the source repository for changes, waits 5 minutes to ensure there aren't more changes pending, then kicks off a full build of the application. All our Unit Tests run after every build using a Test framework like NUnit or MbUnit. We wanted to add testing of PowerShell scripts to our existing Continuous Integration framework. Even though PowerShell is a dynamic scripting environment, that doesn't except it from proper unit testing or first-class inclusion in our build.

This example scripts shows how to create an instance of the PowerShell "Runspace" environment in-process within an existing Unit Testing Framework like NUnit. This example tests a script called "new-securestring.ps1" that returns a .NET Framework 2.0 SecureString type. The object is returned from the PowerShell script and passed back for testing. Then the SecureString is added to the PowerShell environment as a named variable that is accessible from another PowerShell script, completing the circle.

```
using System;
using System.Collections;
using System.Collections.ObjectModel;
using System.Management.Automation.Runspaces;
using System.Management.Automation;
using NUnit.Framework;
using System.Security;
namespace PSUnitTestLibrary.Test
{
    [TestFixture]
    public class Program
    {
        private Runspace myRunSpace;

        [TestFixtureSetUp]
        public void PSSetup()
        {
            myRunSpace = RunspaceFactory.CreateRunspace();
            myRunSpace.Open();
            Pipeline cmd = myRunSpace.CreatePipeline(@"set-Location
'C:\dev\someproject");
            cmd.Invoke();
         }
        [Test]
        public void PSTest()
        {
            Pipeline cmd = myRunSpace.CreatePipeline("get-location");
            Collection<PSObject> resultObject = cmd.Invoke();
            string currDir = resultObject[0].ToString();
            Assert.IsTrue(currDir == @"'C:\dev\someproject");
            cmd = myRunSpace.CreatePipeline(@".\new-securestring.ps1
password");
            resultObject = cmd.Invoke();
            SecureString ss =
(SecureString)resultObject[0].ImmediateBaseObject;
            Assert.IsTrue(ss.Length == 8);
            myRunSpace.SessionStateProxy.SetVariable("ss", ss);
            cmd = myRunSpace.CreatePipeline(@".\getfrom-securestring.ps1
$ss");
            resultObject = cmd.Invoke();
            string clearText = (string)resultObject[0].ImmediateBaseObject;
            Assert.IsTrue(clearText == "password");
        }
    }
}
```

In this case we're assuming one PowerShell RunSpace per TestFixture as we are using the TestFixtureSetUp method to get that RunSpace going, but one could certainly move things around if you wanted different behavior or isolation. The PowerShell RunspaceFactory allows for as many RunSpaces as are required, so tests can scope them as appropriate.

Including PowerShell into our existing Unit Testing Framework ensured that the dynamic and flexible no-compile model of PowerShell didn't encourage sloppiness. In this model, every line of PowerShell code is subject to the same scrutiny as compiled C# code.

## Deploying Software

We've semantically and physically separated the concept of *deployment* from *publishing*. We store our applications in their deployed or "pre-installed" state in Subversion so that they can be retrieved on a fresh machine and run immediately. However, there are some post-deployment steps like registry settings, COM object registration and ACL permission that need to be setup after a deployment.

The commands to deploy a complete Consumer (Retail) Banking System on a Virtual Machine from scratch on a fresh operating system with PowerShell follow:

```
PS:> setup-voyager.ps1 –role "ALL"
PS:> add-fi.ps1 –fiName "SampleBank"
PS:> deploy-fi.ps1 –fiName "SampleBank"
PS:> test.ps1 –fiName "SampleBank"
```

The first line runs a script that brings Voyager down from the Subversion repository and uses PowerShell to register performance counters, register COM objects, edit XML configuration files, etc. The outer shell for that script looks like this:

```
# Executes the setup steps common across all server roles
function
InstallCommon([string]$Path,[string]$voyUser,[string]$voyPwd,[string]$SQLSer
verName,[string]$RepoPath,[bool]$ccexec)
{
      write-verbose "RepoPath: $RepoPath"
      Checkout -RepoPath $RepoPath -Path $Path

      LoadAssemblies
      CopySystemFiles -Path $Path
      AddRegistryKeys -Path $Path
      SetUpSecretStore -Path $Path -fileList $ssfiles
      SetupSDK -Path $Path -fileList $sdkfiles
      InstallPerfCounters -Path $Path
      SetupDCOM -voyUser $voyUser -voyPwd $voyPwd
      AddUserToDCOMUsersGroup -voyUser $voyUser
      SetWMIPermissions -voyUser $voyUser
      SetTempPermissions -voyUser $voyUser
      SetupGAC -Path $Path
}
```

We found a comfortable balance between C# and PowerShell during this project. PowerShell scripts became the interstitial bits, the string, that holds the major reusable components together. When something was easier to accomplish in PowerShell, we wrote scripts. When it was clear that a more strongly typed language was required, we wrote Assemblies or cmdlets. It's very easy to move back and forth between the two environments.

Even though PowerShell is its own language, we found that engineers and QA that had a background in C# had no trouble picking it up. Some of the syntax is a little obscure, but good design patterns still apply and our engineers were able to apply common patterns of reuse to their work in PowerShell.

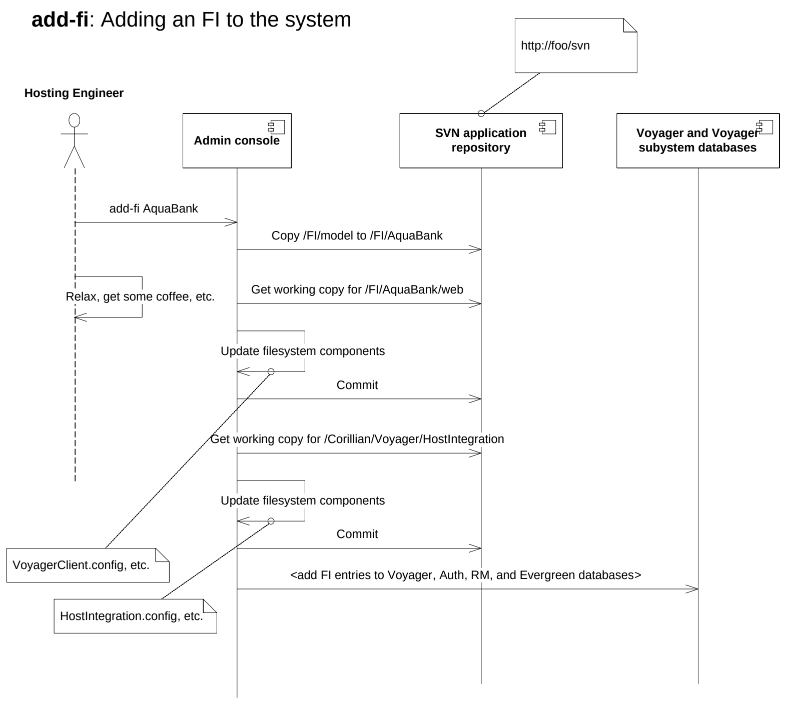

add-fi: Adding an FI to the system
http://foo/svn
Hosting Engineer
Admin console
SVN application repository
Voyager and Voyager subystem databases
add-fi AquaBank
Copy /FI/model to /FI/AquaBank
Relax, get some coffee, etc.
Get working copy for /FI/AquaBank/web
Update filesystem components
Commit
Get working copy for /Corillian/Voyager/HostIntegration
Update filesystem components
Commit
VoyagerClient.config, etc.
<add FI entries to Voyager, Auth, RM, and Evergreen databases>
HostIntegration.config, etc.

Figure 6 - Sequence Diagram showing how an FI is added to the system

Other scripts install the Web Server, SQL Server and other support servers. The next script, Add-FI registers the new Financial Institution with the Voyager system and brings the Web Application down from Subversion and applies the default settings. The last line, test.ps1, runs an extensive series of Unit and Integration Tests to confirm the successful deployment.

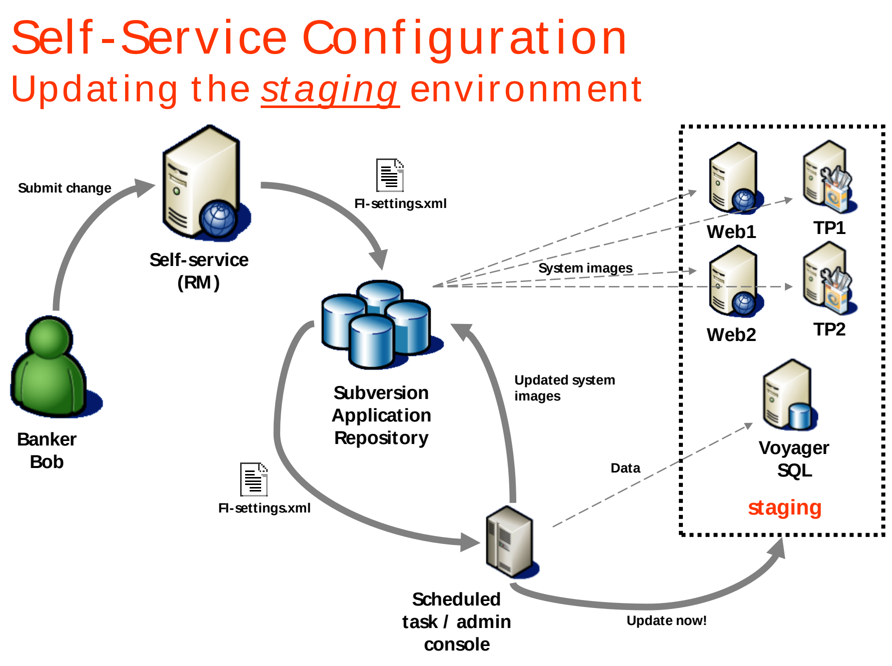
The pod includes a "MODEL" for each application that might need to be installed. A Financial Institution who signs up with Corillian will select the applications they want from our suite of apps. Each application has a default state that includes the minimum settings it requires to be usable. This allows our sales and demo staff to literally type in these four commands – that could easily be one command – to bring up a sample site that talks to a mainframe simulator.

As there are a number of different kinds of applications that might be deployed, we abstracted the work into Deploy Handlers that know how to handle their respective application type. Each is written in C# and implements a Deploy method and is called by PowerShell when it comes time to deploy the application. In essence, Deploy Handlers know how to "lay the bits down on disk" and little else.

## Publishing Configuration Settings

Once a Financial Institution is deployed – meaning the files are laid down in a default state on the disk and a minimal configuration is prepared - the next step is publishing their settings. The *publish* step occurs every time a bank makes a change to their settings from their web-based configuration console.

Figure 7 - Slide showing settings moving from Subversion to a Staging



### System

In a pattern similar to deployment, each class of application has an associated PublishHandler. The publish handler is responsible for taking the FI-specific settings from a "settings bag" and applying those settings to the class of object the handler is responsible for. For example, some PublishHandlers are DatabasePublishHandler, UIResourcePublishHandler, UIStylePublishHandler, and WebSettingsPublishHandler.

As a more specific example, we allow the client to change the fonts and colors of their site, so the UIStylePublishHandler takes the name/value pair configuration details like “HeaderFont” = “Ariel” as well as dozens of others, and then writes out a custom CSS file dynamically. The same pattern applies to changing text on the site, image assets – anything we choose to make changeable to the client.

## The Missing Link - PowerShell Remoting

PowerShell doesn’t include a technique to issue commands to remote systems in version 1.0, so we had to build our own for now. This is a very desirable feature and I expect to see something similar included in a future version of PowerShell.

The basic requirement is to issue PowerShell commands to many different machines in a distributed fashion. After some pair programming with Corillian Architect Brian Windheim, we created a Windows Service that would take a string of PowerShell commands and return a string that was the console output of those commands. We could then issue remote commands, but the CLR type passed to and from at the server was just strings. We were using PowerShell but we’re just made the equivalent of SysInternal’s PSEXEC[3] utilty, only for PowerShell. We preferred something more integrated with the PowerShell pipeline. Specifically we wanted type fidelity of return values.

Ideally we’d like to have behavior like this, but again, PowerShell 1.0 doesn’t include this:

```
using (Runspace myRunSpace = RunspaceFactory.CreateRunspace("COMPUTERNAME"))
{
    myRunSpace.Open();
}
```

We then realized that we could use the built-in PowerShell cmd-let called Export-CliXml[4]. It is the *public* cmdlet that serializes CLR and PowerShell objects to a custom XML format. It’s important to note that it isn't the XmlSerializer. It's a serialized graph of objects with a rich enough description of those objects that the client doesn't necessarily need the CLR types. If reflection had a serialization format, it might look like this CLI-XML format.

We created a RunspaceInvoker class that would be hosted in a Windows Service or IIS on each remote machine, but it could be in any Remoting hosting process. We host in IIS using .NET 2.0 in order to use the built in Windows Integrated Security over remoting. The app.config for my service looks like this:

```
<?xml version="1.0"  encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.runtime.remoting>
    <customErrors mode="Off"/>
    <application>
      <channels>
        <channel ref="http" port="8081"/>
      </channels>
```

[3] http://www.sysinternals.com/Utilities/PsExec.html
[4] https://www.microsoft.com/technet/scriptcenter/topics/msh/cmdlets/export-clixml.mspx

```
      <service>
        <wellknown mode="SingleCall"
                   type="Example.RemoteRunspace.RunspaceInvoker,
                   Example.RemoteRunspace" objectUri="remoterunspace.rem"/>
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Note the objectUri and port, they are used for the endpoint address. There's an installer class that is run using installutil.exe on each destination machine. You can either set the identity of a Windows Service and starts it up with `net start RemoteRunspaceService`, or you can host within IIS and manage process identity the standard way.

This is the RunspaceInvoker, it's very simple. The error handling has been removed for clarity.

```
 public class RunspaceInvoker : MarshalByRefObject
 {
    public RunspaceInvoker(){}

    public string InvokeScriptBlock(string scriptString)
    {
        using (Runspace myRunSpace = RunspaceFactory.CreateRunspace())
        {
            myRunSpace.Open();

            string tempFileName = System.IO.Path.GetTempFileName();
            string newCommand = scriptString +
                " | export-clixml " + "\"" + tempFileName + "\"";
            Pipeline cmd = myRunSpace.CreatePipeline(newCommand);

            Collection<PSObject> objectRetVal = cmd.Invoke();

            myRunSpace.Close();

            string retVal = System.IO.File.ReadAllText(tempFileName);
            System.IO.File.Delete(tempFileName);
            return retVal;
        }
    }
 }
```

A command for the remote service comes into the scriptString parameter. For example we might pass in `dir c:\temp` as the string, or a whole long pipeline. We create a Runspace, open it and append "`| export-clixml`" and put the results in a tempFileName.

It's unfortunate we can't put the pipeline results in a variable or get it out of the Pipeline, but I think I understand why they force me to write the CLI-XML to a file. They are smuggling the information out of the system. It's the *Heisenberg Uncertainly Principle of PowerShell*. If you observe something, you change it. Writing the results to a file is a trapdoor that doesn't affect the output of the pipeline. Again, this will likely be a moot point in future versions. We've tried to abstract things away so that when a future is added in a later version, we'll only need to remove our custom code.

This remoting doesn't need to be highly performant as it’s only happening during configuration or deployment. The pipeline results are written to a temp file, we read the file in then delete it immediately. The serialized CLI-XML is returned to the caller.

The client portion includes two parts, a RunspaceProxy, and a Type Extension. We start with the RunspaceProxy. This is the class that the client uses to invoke the command remotely.

```
public class RunspaceProxy
{
    public RunspaceProxy()
    {
        HttpChannel chan = new HttpChannel();
        if (ChannelServices.GetChannel("http") != null)
        {
            ChannelServices.RegisterChannel(chan, false);
        }
     }

    public Collection<PSObject> Execute(string command, string remoteurl)
    {
        RunspaceInvoker proxy = (RunspaceInvoker)Activator.GetObject(
              typeof(RunspaceInvoker), remoteurl);
        string stringRetVal = proxy.InvokeScriptBlock(command);

        using (Runspace myRunSpace = RunspaceFactory.CreateRunspace())
        {
            myRunSpace.Open();
            string tempFileName = System.IO.Path.GetTempFileName();
            System.IO.File.WriteAllText(tempFileName, stringRetVal);
            Pipeline cmd = myRunSpace.CreatePipeline(
             "import-clixml " + "\"" + tempFileName + "\"");
              Collection<PSObject> retVal = cmd.Invoke();
              System.IO.File.Delete(tempFileName);
              return retVal;
        }
    }
}
```

We use the HTTP channel for debugging and ease of use with tools like TcpTrace[5]. The command to be executed comes in along with the remoteUrl. We instantiate a RunspaceInvoker on the *remote machine* and it does the work via a call to InvokeScriptBlock in a hosted Runspace. The exported CLI-XML comes back over the wire and now I have to make a tempfile on the client. Then, in order to 'deserialize' - a better word would might be *re-hydrate* - the Collection of PSObjects, make a local Runspace and call import-clixml and poof, a Collection<PSObject> is returned to the client. Then the file is deleted.

Why is returning real PSObjects so important when the first version worked fine returning strings? Because when a script or cmdlet returns a PSObject we can use the `select`, `sort`, and `where` cmdlets against these PSObjects as if they were locally created – because they are local. They are real

[5] http://www.pocketsoap.com/

and substantial. This will allow us to write scripts that blur the line between the local admin and remote admin.

Now, all of these samples have been C# so far, when does PowerShell come in? Also, since we've worked so hard to get the return values integrated cleanly with PowerShell, what's a good way to get the remote *calling* of scripts integrated cleanly?

Our first try was to simple make a global function called `RemoteInvoke()` that took a command string and returned an object. It worked, but didn't feel well-integrated. While reading then PowerShell blog, we remembered how Jeffrey Snover said to look to Type Extensions when adding functionality[6] rather than functions and cmdlets.

A global function is just an expression of programmer intent that is floating around in the global environment. We wanted to actually take a piece a functionality that already worked well, the ScriptBlock, and extend it. In traditional object-oriented systems extension is done via derivation, but in PowerShell (as well as languages like Ruby) we have *type extension* as an available option. That means we can literally add functionality, in this case a new method called RemoteInvoke, to an existing type. Type extension allows for a tighter integration with PowerShell, doesn't change the usage model dramatically, and makes the new functionality not only easier to learn, but also more discoverable.

We made a My.Types.ps1xml file in the  PSConfiguration directory with our function enhancing the ScriptBlock type within PowerShell:

```
<Types>
  <Type>
    <Name>System.Management.Automation.ScriptBlock</Name>
    <Members>
      <ScriptMethod>
        <Name>RemoteInvoke</Name>
        <Script>
         if ($args[0])
         {
             $PRIVATE:remoteUrl = $args[0]
         }
         else
         {
             $PRIVATE:remoteUrl = $GLOBAL:remoteUrl
         }
                if ($PRIVATE:remoteUrl -eq $null) { throw 'Please supply a
remoteUrl either by global variable or argument!' }
                if ($GLOBAL:evergreenlibPath -eq $null) { throw 'The
Evergreen Environment is not setup!' }

[System.reflection.assembly]::LoadWithPartialName("System.Runtime.Remoting")
|
               out-null
          $someDll = "C:\foo\Hanselman.RemoteRunspace.dll"
          $asm = [System.Reflection.Assembly]::LoadFrom($someDll) | out-null
```

6 http://blogs.msdn.com/powershell/archive/2006/06/24/644987.aspx

```
          $runspace = new-object Hanselman.RemoteRunspace.RunspaceProxy

          $runspace.Execute([string]$this, $GLOBAL:remoteUrl);
        </Script>
      </ScriptMethod>
    </Members>
  </Type>
</Types>
```

A call to `Update-TypeData My.Types.ps1xml` happens within the profile so it happens automatically. This file adds a new method to the built-in ScriptBlock type. A ScriptBlock is literally a block of script within curly braces. It's a very natural "atom" for us to use in PowerShell.

The RemoteUrl is an optional parameter to the RemoteInvoke ScriptMethod, and if it's not passed in, we'll fall back to a global variable. The RemoteInvoke loads the .NET System.Runtime.Remoting assembly, and then it loads our Proxy assembly. Then it calls Execute, casting the [ScriptBlock] to a [string] because the Runspace takes a string.

For example, at a PowerShell prompt if we do this:

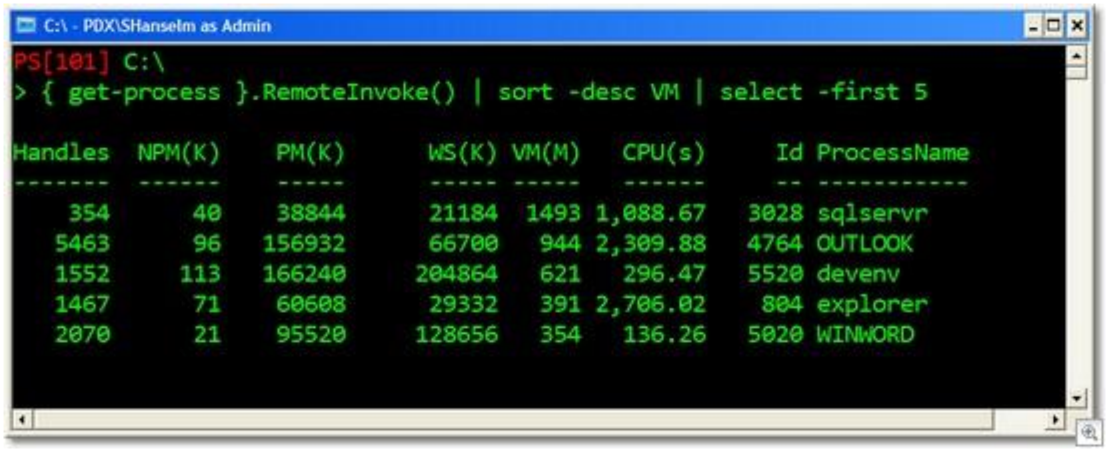
```
PS[79] C:\> $remoteUrl="http://remotecomputer:8081/RemoteRunspace.rem"

PS[80] C:\PS[80] C:\> 2+2
4

PS[81] C:\> (2+2).GetType()
IsPublic IsSerial Name    BaseType
-------- -------- ----    --------
True     True     Int32   System.ValueType

PS[82] C:\> {2+2}.GetType()
IsPublic IsSerial Name          BaseType
-------- -------- ----          --------
True     False    ScriptBlock   System.Object

PS[83] C:\> {2+2}
4

PS[84] C:\> {2+2}.RemoteInvoke()
4

PS[85] C:\> {2+2}.RemoteInvoke().GetType()

IsPublic IsSerial Name    BaseType
-------- -------- ----    --------
True     True     Int32   System.ValueType
```

Note the result of the last line. The value that comes out of RemoteInvoke is an Int32, not a string. The result of that ScriptBlock executing is a PowerShell type that we can work with elsewhere in a local script.

Here's the CLI-XML that went over the wire (just to make it clear it's not XmlSerializer XML):

```
<Objs Version="1.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <I32>4</I32>
</Objs>
```

The call to 2+2 is a simple example, but this technique works with even large and complex object graphs like the FileInfos and FileSystemInfo objects that are returned from dir (get-childitem) as seen in the Figure below.



```
C:\ - PDX\SHanselm as Admin
PS[101] C:\
> { get-process }.RemoteInvoke() | sort -desc VM | select -first 5

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    354      40    38844      21184  1493 1,088.67   3028 sqlservr
   5463      96   156932      66700   944 2,309.88   4764 OUTLOOK
   1552     113   166240     204864   621   296.47   5520 devenv
   1467      71    60608      29332   391 2,706.02    804 explorer
   2070      21    95520     128656   354   136.26   5020 WINWORD
```

Figure 8 - A call to Get-Process that executed on a Remote Machine continued along the pipeline

In this screenshot we do a `get-process` on the remote machine then sort and filter the results using standard cmdlets just as we would/could if the call were local.

## Distributed Deployment - Background Job Processing with PowerShell

Now that we can issue commands remotely, the next step is issuing those commands asynchronously. Jim Truher[7] used the PowerShell RunSpace architecture to create the next important element of our system – background processing. When managing large web farms, the more servers, the more important asynchrony becomes. Early versions of our system were tested in a one or two server configuration with an algorithm like this psuedocode:

```
for each Server in Pod.Servers
    for each Role in Server.Roles
        Remote Deploy Role-specific code to Server
    next
next
```

However, when the system was used in larger farms it becomes painfully obvious that the Servers are getting their code deployed one at a time, in order.

Windows PowerShell doesn't natively support the concept of background jobs, but the ability to fire up a new RunSpace makes the implementation of Un*x style jobs processing a fairly trivial task.

The experience at the command line looks like this:

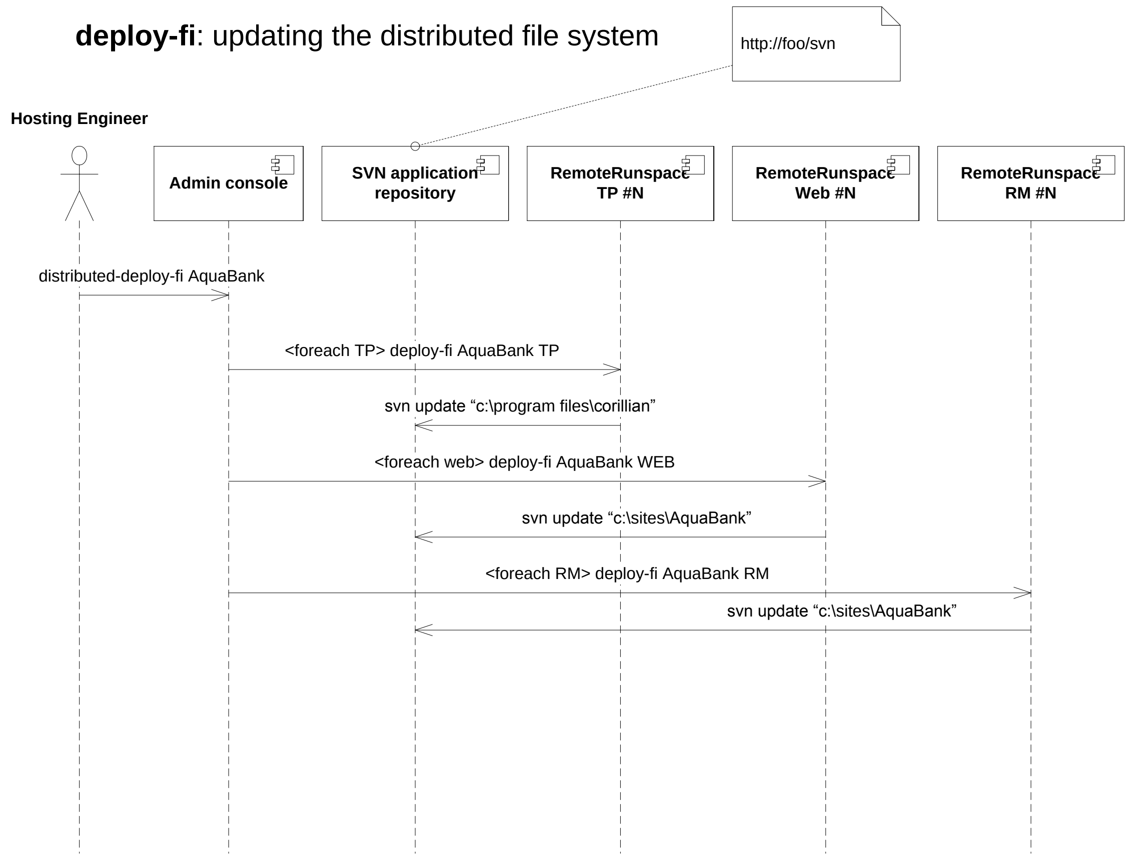
[7] http://jtruher.spaces.live.com/blog/cns!7143DA6E51A2628D!130.entry

```
PS> new-job { get-date; start-sleep 5; get-date }
Job 0 Started

PS>

Job 0 Completed

PS> jobs 0
JobId    Status       Command                       Results
-----    ------       -------                       -------
5        Completed    get-date; start-sleep 5; ... 11/30/2006 8:27:32 PM
PS> (jobs 0).results
Thursday, November 30, 2006 8:27:32 PM
Thursday, August 30, 2006 8:27:37 PM
```

This allows us to change the original deployment psuedocode to:

```
for each Server in Pod.Servers
    for each Role in Server.Roles
        new-job Remote Deploy Role-specific code to Server
    next
next
wait while all jobs are still running
```

This simple change makes a huge difference in the performance of a deployment through parallelization, but doesn't require changes to any of the underlying code. ScriptBlocks can now be executed as local background jobs with execution occurring on remote machines.  Figure 9 shows a sequence diagram with the steps required to deploy a Bank, in this example "Aquabank", to a farm of servers, where #N is the number of servers in the farm.



deploy-fi: updating the distributed file system
http://foo/svn
Hosting Engineer
Admin console
SVN application repository
RemoteRunspace TP #N
RemoteRunspace Web #N
RemoteRunspace RM #N
distributed-deploy-fi AquaBank
<foreach TP> deploy-fi AquaBank TP
svn update "c:\program files\corillian"
<foreach web> deploy-fi AquaBank WEB
svn update "c:\sites\AquaBank"
<foreach RM> deploy-fi AquaBank RM
svn update "c:\sites\AquaBank"

Figure 9 - Sequence Diagram showing the Distributed Deployment of an Application

## Securing PowerShell Scripts

The RemoteRunspace will be installed on production machines, but will be locked down in a number of ways. First, the RemoteRunspace will listen only on an “administrative backplane,” that is, a network that is not the public internet. Second, it will use SSPI or Windows Authentication over .NET 2.0 Remoting for Authentication. Thirdly, we will explicitly disallow running of any arbitrary PowerShel scripts by changing the execution policy of PowerShell to “AllSigned” and force certificate signing of all scripts.

PowerShell supports a concept called "execution policies" in order to help deliver a more secure command line administration experience.  Execution policies define the restrictions under which PowerShell loads files for execution and configuration.  The four execution policies are Restricted, AllSigned, RemoteSigned, and Unrestricted.

PowerShell is configured to run in its most secure mode by default.  It installed in this mode with a "Restricted" execution policy, where PowerShell operates as an interactive shell only.

The modes are:

- **Restricted** (default execution policy, does not run scripts, interactive only)
- **AllSigned** (runs scripts; all scripts and configuration files must be signed by a publisher that you trust; opens you to the risk of running signed (but malicious) scripts, after confirming that you trust the publisher);
- **RemoteSigned** (runs scripts; all scripts and configuration files downloaded from communication applications such as Microsoft Outlook, Internet Explorer, Outlook Express and Windows Messenger must be signed by a publisher that you trust; opens you to the risk of running malicious scripts not downloaded from these applications, without prompting)
- **Unrestricted** (runs scripts; all scripts and configuration files downloaded from communication applications such as Microsoft Outlook, Internet Explorer, Outlook Express and Windows Messenger run after confirming that you understand the file originated from the Internet; no digital signature is required; opens you to the risk of running unsigned, malicious scripts downloaded from these applications).

### Restricted Execution Policy

If you're reading this for the first time, PowerShell may have just displayed the error message as you tried to run a script:

```
The file C:\my_script.ps1 cannot be loaded. The execution of scripts is
disabled on this system. Please see "Get-Help about_signing" for more
details.
```

By default, PowerShell does not run scripts, and loads only configuration files signed by a publisher that you trust. Run the following from a PowerShell prompt (AllSigned is an example):

```
Set-ExecutionPolicy AllSigned
```

This command requires administrator privileges.  Changes to the execution policy are recognized immediately.The AllSigned execution policy is best for production since it forces the requirement for digital signatures on *all* scripts and configuration files.

### Script Signing Background

Adding a digital signature to a script requires that it be signed with a code signing certificate.  Two types are suitable: those created by a certificate authority (such as Verisign etc.), and those created by a user (called a self-signed certificate).

If your scripts are specific to your internal use, you maybe able to self-sign. You can also buy a code signing certificate from another certificate authority if you like.

For a self-signed certificate, a designated computer is the authority that creates the certificate.  The benefits of self-signing include its zero cost as well as creation speed and convenience.  The drawback is that the certificate must be installed on every computer that will be running the scripts, since other computers will not trust the computer used to create the certificate.

To create a self-signed certificate, the makecert.exe program is required from the Microsoft .NET Framework SDK or Microsoft Windows Platform SDK. It's found in the "C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin\" directory.

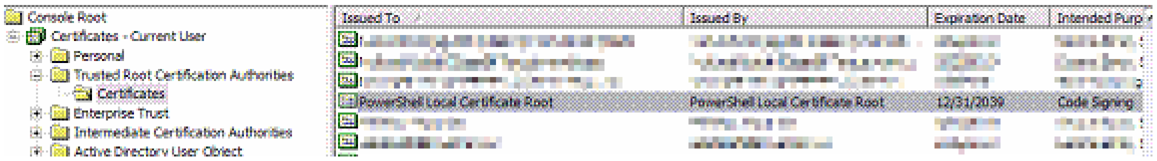
Set up to view the Certificates by running mmc.exe and adding the Certificates snap-in, or by running certmgr.msc from Start|Run.

## Setting Up a Self-Signed Certificate

Run the following from a Command Prompt. It creates a local certificate authority for your computer:

```
makecert -n "CN=PowerShell Local Certificate Root" -a sha1 -eku
1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer -ss Root -sr localMachine
```

You will be prompted for the private key twice, this will create the trusted root certificate authority:



- Console Root
  - Certificates - Current User
    - Personal
    - Trusted Root Certification Authorities
      - Certificates
    - Enterprise Trust
    - Intermediate Certification Authorities
    - Active Directory User Object

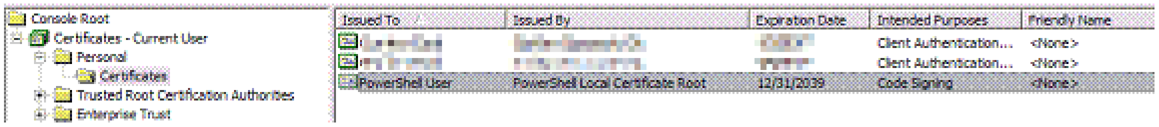
| Issued To | Issued By | Expiration Date | Intended Purp |
|---|---|---|---|
| [redacted] | [redacted] | [redacted] | [redacted] |
| [redacted] | [redacted] | [redacted] | [redacted] |
| [redacted] | [redacted] | [redacted] | [redacted] |
| PowerShell Local Certificate Root | PowerShell Local Certificate Root | 12/31/2039 | Code Signing |
| [redacted] | [redacted] | [redacted] | [redacted] |
| [redacted] | [redacted] | [redacted] | [redacted] |

Figure 10 - A new trusted root certificate authority

Now run the following from a Command Prompt. It generates a personal certificate from the above certificate authority:

```
makecert -pe -n "CN=PowerShell User" -ss MY -a sha1 -eku 1.3.6.1.5.5.7.3.3 -
iv root.pvk -ic root.cer
```

You'll be prompted for the private key, there will now be a certificate in the Personal store:



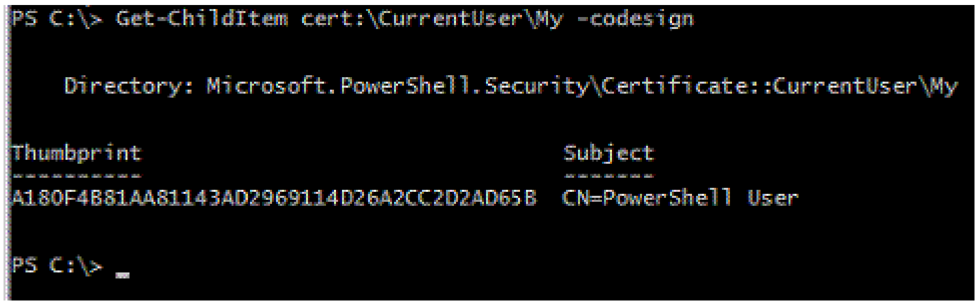
- Console Root
  - Certificates - Current User
    - Personal
      - Certificates
    - Trusted Root Certification Authorities
    - Enterprise Trust

| Issued To | Issued By | Expiration Date | Intended Purposes | Friendly Name |
|---|---|---|---|---|
| [redacted] | [redacted] | [redacted] | Client Authentication... | <None> |
| [redacted] | [redacted] | [redacted] | Client Authentication... | <None> |
| PowerShell User | PowerShell Local Certificate Root | 12/31/2039 | Code Signing | <None> |

Figure 11 - A new certificate in the Personal Store

After the above steps, verify from within Powershell that the certificate was generated correctly:

```
PS C:\ > Get-ChildItem cert:\CurrentUser\My -codesign
```



```
PS C:\> Get-ChildItem cert:\CurrentUser\My -codesign

    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                Subject
----------                                -------
A180F4B81AA81143AD2969114D26A2CC2D2AD65B  CN=PowerShell User

PS C:\> _
```

Figure 12 - Accessing the certificate store from PowerShell

You can now delete the two temporary files root.pvk and root.cer in your working directory. The certificate info is stored with that of others, in "C:\Documents and Settings\[username]\Application Data\Microsoft\SystemCertificates\My\".

## Signing a Script

To test the effectiveness of digitally signing a Powershell script, try it with a script "foo.ps1":

```
PS C:\> Set-ExecutionPolicy AllSigned
PS C:\> .\foo.ps1
The file C:\foo.ps1 cannot be loaded. The file C:\foo.ps1 is not digitally
signed. The script will not execute on the system. Please see "get-help
about_signing" for more details..
At line:1 char:9
+ .\foo.ps1 <<<<
```

Now sign the script:

```
PS C:\> Set-AuthenticodeSignature c:\foo.ps1 @(Get-ChildItem
cert:\CurrentUser\My -codesigning)[0]
Directory: C:\
SignerCertificate                         Status             Path
-----------------                         ------             ----
A180F4B81AA81143AD2969114D26A2CC2D2AD65B  Valid              foo.ps1
```

This modifies the end of the script with a signature block comment at its end. For example, if the script consisted of the following commands:

```
param ( [string] $You = $(read-host "Enter your first name") )
write-host "$This was signed."
```

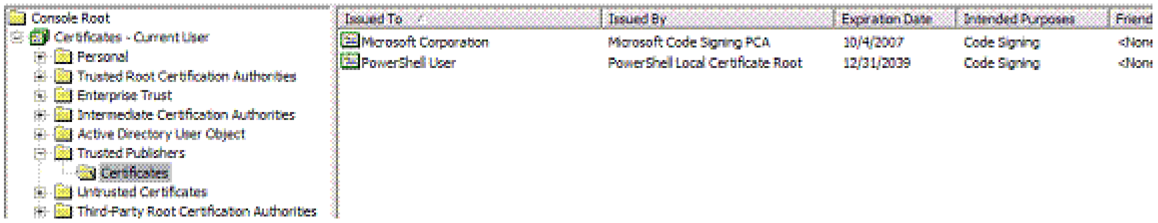
After the script is signed, it looks like this:

```
param ( [string] $You = $(read-host "Enter your first name") )
write-host "$ This was signed."
# SIG # Begin signature block
# MIIEMwYJKoZIhvcNAQcCoIIEJDCCBCACAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQU6vQAn5sf2qIxQqwWUDwTZnJj
...snip...
# m5ugggI9MIICOTCCAaagAwIBAgIQyLeyGZcGA4ZOGqK7VF45GDAJBgUrDgMCHQUA
# Dxoj+2keS9sRR6XPl/ASs68LeF8o9cM=
# SIG # End signature block
```

Execute the script once again:

```
PS C:\> .\foo.ps1
Do you want to run software from this untrusted publisher?
The file C:\foo.ps1 is published by CN=PowerShell User. This publisher is
not trusted on your system. Only run scripts from trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help
(default is "D"):
```

Answer "A" and the script proceeds to run, and runs without prompting thereafter. A new certificate is also created in the Trusted Publishers container:



- Console Root
  - Certificates - Current User
    - Personal
    - Trusted Root Certification Authorities
    - Enterprise Trust
    - Intermediate Certification Authorities
    - Active Directory User Object
    - Trusted Publishers
      - Certificates
    - Untrusted Certificates
    - Third-Party Root Certification Authorities

| Issued To | Issued By | Expiration Date | Intended Purposes | Friend |
|---|---|---|---|---|
| Microsoft Corporation | Microsoft Code Signing PCA | 10/4/2007 | Code Signing | <None |
| PowerShell User | PowerShell Local Certificate Root | 12/31/2039 | Code Signing | <None |

Figure 13 - A new certificate in Trusted Publishers

If the certificate is missing the script will fail.

### Running Signed Scripts Elsewhere

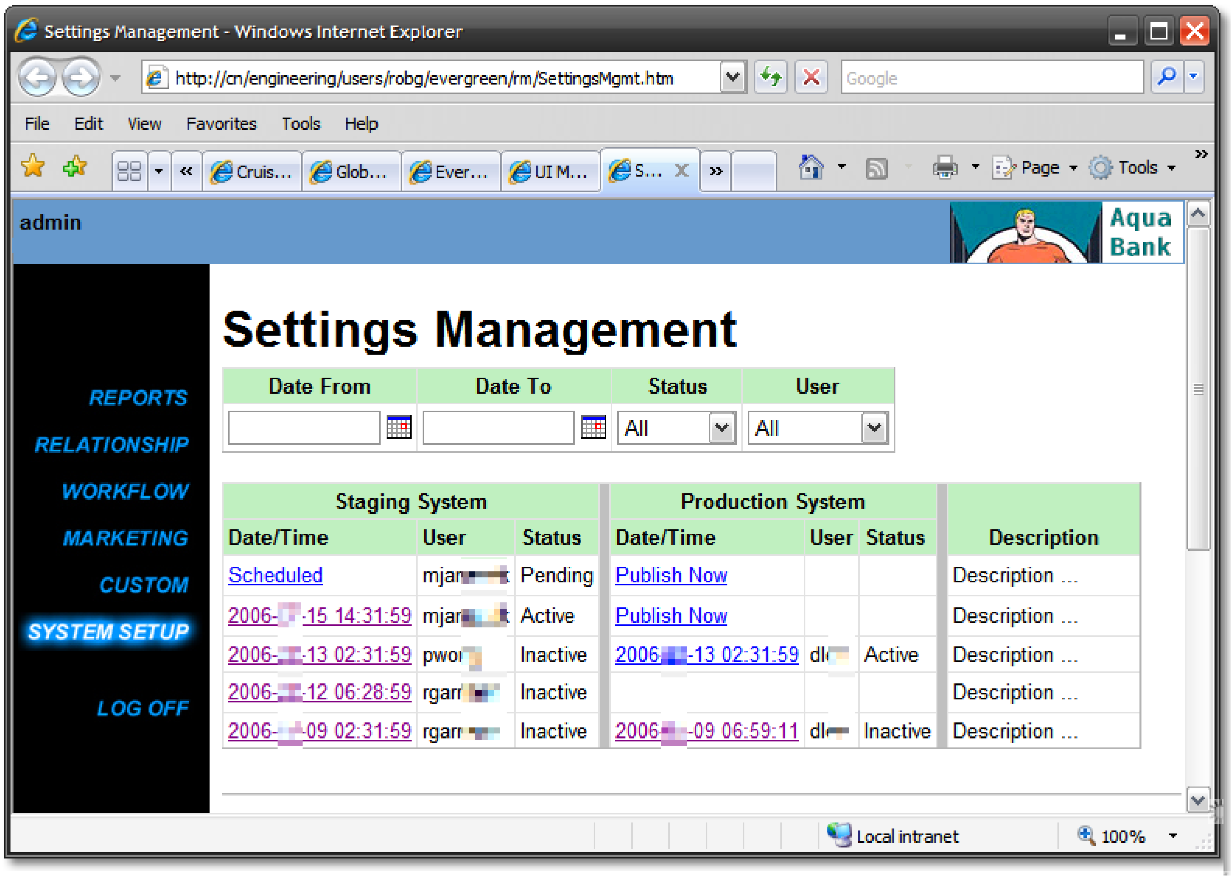
PowerShell will be unable to validate a signed script on computers other than the one where it was signed. Attempting to do so gives an error:

```
PS C:\ > .\foo.ps1
The file C:\foo.ps1 cannot be loaded. The signature of the certificate can
not be verified.
At line:1 char:9
+ .\foo.ps1 <<<<
```

Signed scripts can be transported by exporting (from original computer) and importing (to the new computer) the Powershell certificates found in the Trusted Root Certification Authorities container. Optionally, the Trusted Publishers can also be moved to prevent the first-time prompt.

## Letting the Customer Manage Configuration

The entire solution needs to be a breeze to manage and administer by non-IT-savvy individuals, so along with the PowerShell interface, there is a Web-based interface for changing settings and publishing them from development to staging, then to production.


Settings Management - Windows Internet Explorer
http://cn/engineering/users/robg/evergreen/rm/SettingsMgmt.htm
Google
File Edit View Favorites Tools Help
Cruis… Glob… Ever… UI M… S…
Page Tools
Aqua Bank
REPORTS
RELATIONSHIP
WORKFLOW
MARKETING
CUSTOM
SYSTEM SETUP
LOG OFF
Local intranet
100%

admin

**Settings Management**

| Date From | Date To | Status | User |
|---|---|---|---|
| | | All | All |

| Staging System | | | Production System | | | |
|---|---|---|---|---|---|---|
| Date/Time | User | Status | Date/Time | User | Status | Description |
| Scheduled | mjar[redacted] | Pending | Publish Now | | | Description ... |
| 2006-[redacted]-15 14:31:59 | mjar[redacted] | Active | Publish Now | | | Description ... |
| 2006-[redacted]-13 02:31:59 | pwo[redacted] | Inactive | 2006[redacted]-13 02:31:59 | dl[redacted] | Active | Description ... |
| 2006-[redacted]-12 06:28:59 | rgar[redacted] | Inactive | | | | Description ... |
| 2006-[redacted]-09 02:31:59 | rgar[redacted] | Inactive | 2006[redacted]-09 06:59:11 | dl[redacted] | Inactive | Description ... |

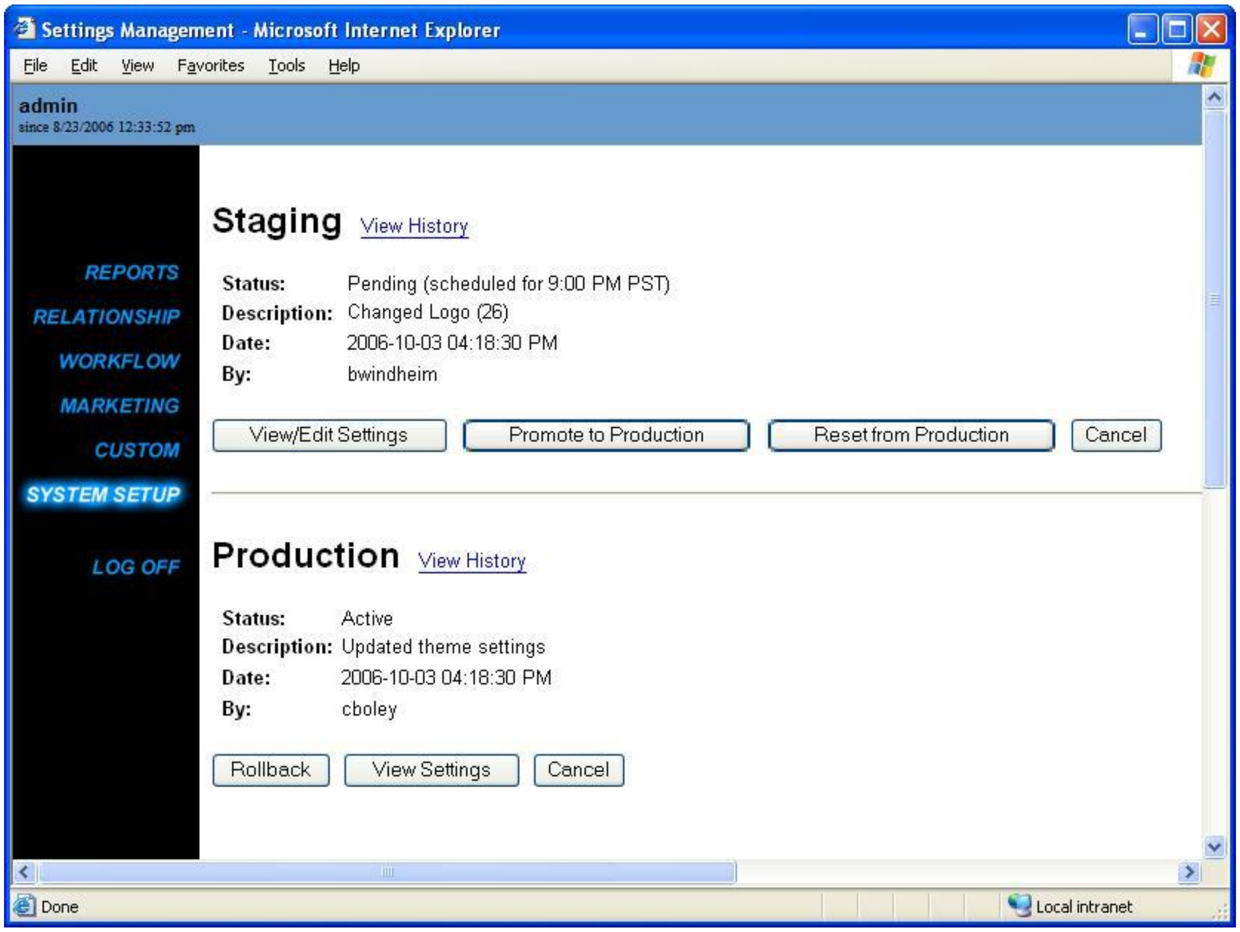
Figure 14 - UI for managing settings between Staging and Production

Most importantly, because we use Subversion as our versioned file system, every change is audited and tagged with Subversion so that the state of the system – every app and every setting – can not only be checked against the version under source control, but changes can be rolled back or settings can be reapplied.

All the PowerShell scripts and cmdlets can be called from within ASP.NET with code like this, using the Runspace model as seen in the NUnit example before. Note that we can add our custom cmdlets by calling AddPSSnapIn and passing the RunspaceConfiguration instance to the RunspaceFactory.

```
private Collection<PSObject> ExecuteCmdLet(Command cmdLet)
{
    PSSnapInException warning;
    Collection<PSObject> result;

    RunspaceConfiguration rsConfig = RunspaceConfiguration.Create();
    rsConfig.AddPSSnapIn("EvergreenAdministration", out warning);

    using (Runspace rs = RunspaceFactory.CreateRunspace(rsConfig))
    {
        rs.Open();

        using (Pipeline p = rs.CreatePipeline())
        {
            p.Commands.Add(cmdLet);
            result = p.Invoke();
        }
        rs.Close();
    }
    return result;
}
```

When calling a PowerShell command from a “host” application like ASP.NET you should avoid concatenating strings to prevent “SQL-injection style attacks.” We don’t want to allow arbitrary script to be called within our hosted PowerShell Runspace, so one should treat commands in PowerShell similarly to stored procedures in SQL.



admin
since 8/23/2006 12:33:52 pm

Staging

Status: Pending (scheduled for 9:00 PM PST)
Description: Changed Logo (26)
Date: 2006-10-03 04:18:30 PM
By: bwindheim

Production

Status: Active
Description: Updated theme settings
Date: 2006-10-03 04:18:30 PM
By: cboley

Figure 15 - UI showing Rollback and Promote to Production

Here is an example script from an ASP.NET page that rolls back settings as seen in the UI in Figure 15.

```
protected void btnRollback_Click(object sender, EventArgs e)
{
    Command cmd;
    Collection<PSObject> result;

    cmd = new Command("Get-DeployHistory");
    cmd.Parameters.Add("FIName", Profile.FI);
    cmd.Parameters.Add("Environment",
Corillian.Evergreen.Environment.Production.Name);
    result = ExecuteCmdLet(cmd);
    if (result.Count <= 1)
        return;

    cmd = new Command("Restore-Settings");
    cmd.Parameters.Add("FIName", Profile.FI);
    cmd.Parameters.Add("EnvironmentName",
Corillian.Evergreen.Environment.Production.Name);
    cmd.Parameters.Add("TaskName", (result[1].BaseObject as Task).Name);
    cmd.Parameters.Add("CommittedBy", Profile.UserName);
    result = ExecuteCmdLet(cmd);

    BindPage();
 }

  private Collection<PSObject> ExecuteCmdLet(Command cmdLet, Runspace rs)
    {
        Collection<PSObject> result;

        using (Pipeline p = rs.CreatePipeline())
        {
            p.Commands.Add(cmdLet);
            result = p.Invoke();
        }

        return result;
    }
```

The same basic technique applies to hosting PowerShell functionality in MMC or within a WinForms application.

## Conclusion

This “hands off” approach to system deployment and on-going maintenance continues to save our Systems Engineers time and effort. PowerShell is an incredibly powerful tool for automation and its basic building blocks are exposed for the developer to exploit. If PowerShell were just a console, it’d be interesting, but not fundamentally compelling. The truly interesting things happen when PowerShell is used like MacGyver, the popular show with the character of the same name of the 80s, used his mind. MacGyver would find himself in a foreign prison with a paperclip, rubber band and pen, and would unfailingly break out of prison with some contraption build with these simple building blocks. In our solution Windows Powershell, a very open platform, along with Open Source software Subversion has enabled us to automate not just the mundane aspects of software deployment and configuration, but also enable our clients to manage their applications in a secure and auditable environment. We believe this solution showcases PowerShell not just as an interesting technical solution, but a compelling business solution.

## About Corillian

Corillian is a premier provider of enterprise software and services for the financial services industry. Empowered with Corillian solutions, some of the world's most visionary financial institutions provide their customers with the tools to manage their finances more effectively.

Built on the Microsoft Windows Server System, the Windows .NET framework, and utilizing XML Web Services, Corillian's solutions are unmatched in reliability and performance, and successfully scale at some of the world's largest financial institutions. Corillian's proven solutions enable financial institutions to deliver innovative services enterprise-wide, across multiple delivery channels and multiple lines of business. The Corillian Voyager platform provides secure and scalable account access and transaction processing for a wide variety of applications built by Corillian's expert software developers, by Corillian-certified partners, and by the in-house development organizations of some of the world's best financial institutions. For more information about Corillian Corporation, visit the company's Web site at http://www.corillian.com. NASDAQ: CORI

## About Scott Hanselman

Scott Hanselman is Chief Architect at the Corillian Corporation, an eFinance enabler. He has thirteen years of experience developing software, in last 6 years with VB.NET and C#. Scott is proud to have been appointed the Microsoft Regional Director for Portland, OR for the last six, years, developing content for, and speaking at Developer Days and the Visual Studio.NET Launches in both Portland and Seattle. Scott was in the top 5% of audience-rated speakers at TechEd 2003, 2005, and 2006 in the US and Europe. He's spoken in Africa, Europe and Asia on Microsoft technologies, and co-authored Professional ASP.NET 2.0 with Bill Evjen and Devin Rader. Scott has also spoken at VSLive, and was made a Microsoft MVP for ASP.NET in 2004 and a Solutions Architecture MVP in 2005. His thoughts on the Zen of .NET, Programming and Web Services are at http://www.computerzen.com.